Deterministic Parallel DPLL

Youssef Hamadi, Said Jabbour, Cédric Piette, Lakhdar Saïs

Journal on Satisfiability, Boolean Modeling and Computation 7 (2011) 127-132

Deterministic Parallel DPLL

SYSTEM DESCRIPTION

Youssef Hamadi youssefh@microsoft.com
Microsoft Research
7 J J Thomson Avenue, Cambridge CB3 0FB, United Kingdom
LIX École Polytechnique, F91128 Palaiseau, France

Said Jabbour jabbour@cril.fr
Cedric Piette piette@cril.fr
Lakhdar Sais sais@cril.fr
Université Lille-Nord de France, Artois
CRIL, CNRS UMR 8188, F-62307 Lens
France

Abstract

Current parallel SAT solvers suffer from a non-deterministic behavior. This is the consequence of their architectures which rely on weak synchronizing in an attempt to maximize performance. This behavior is a clear downside for practitioners, who are used to both runtime and solution reproducibility. In this paper, we propose the first Deterministic Parallel DPLL engine. Our experimental results clearly show that our approach preserves the performance of the parallel portfolio approach while ensuring full reproducibility of the results.

KEYWORDS: *SAT solving, parallelism*

Submitted April 2011; revised July 2011; published November 2011

1. Introduction

Parallel SAT solving has received a lot of attention in the last three years. This comes from several factors like the wide availability of cheap multicore platforms combined with the relative performance stall of sequential solvers. Unfortunately, the demonstrated superiority of parallel SAT solvers comes at the price of non reproducible results in both runtime and reported solutions. This behavior is the consequence of their architectures which rely on weak synchronizing in an attempt to maximize performance.

In this work, we propose a deterministic parallel SAT solver. Its results are fully reproducible, i.e., reproducible parallel exploration of the search space, which includes the same reported model or unsatisfiable proof and runtime. It defines a controlled environment based on a total ordering of solvers' interactions through synchronization barriers. To maximize efficiency, information exchange (conflict-clauses) and check for termination are performed on a regular basis. The frequency of these exchanges greatly influences the performance of our solver. The paper explores the trade off between frequent synchronizing which allows the fast integration of foreign conflict-clauses at the cost of more synchronizing steps, and infrequent synchronizing at the cost of delayed foreign conflict-clauses integration.

Algorithm 1: Deterministic Parallel DPLL

```
   Data: A CNF formula F;
   Result: true if F is satisfiable; false otherwise
1  begin
2      <inParallel, 0 ≤ i < nbCores>
3          answer[i] = search(core_i) ;
4      for (i = 0; i < nbCores; i++) do
5          if (answer[i]! = unknown) then
6              return answer[i];
```

In the last two years, portfolio-based parallel solvers became prominent, and we are not aware of a recently developed divide-and-conquer approach. We believe that these parallel portfolio approaches represent the current state-of-the-art in parallel SAT [1, 2, 3, 4].

2. Deterministic Parallel DPLL

In this section, we present the first deterministic portfolio based parallel SAT solver. As sharing clauses is proven to be important for the efficiency of parallel SAT solving, our goal is to design a deterministic approach while maintaining at the same time clause sharing. To this end, our determinization approach is first based on the introduction of a barrier directive (<barrier>) that represents a synchronization point at which a given thread will wait until all the other threads reach the same point. This barrier is introduced to synchronize both clause sharing between the different computing units and termination detection (Section 2.1). Secondly, to enter the barrier region, a synchronization period for clause sharing is introduced and dynamically adjusted (Section 2.2).

2.1 Static Determinization

Let us now describe our determinization approach of non-deterministic portfolio based parallel SAT solvers. Let us recall that a portfolio based parallel SAT solver runs different incarnations of a DPLL-engine on the same instance. Lines 2 and 3 of the Algorithm 1 illustrate this portfolio aspect by running in parallel these different search engines on the available cores. To avoid non determinism in term of a reported solution or an unsatisfiability proof, a global data structure called *answer* is used to record the satisfiability answer of these different cores. The different threads or cores are ordered according to their threads ID (from 0 to nbCores-1). Algorithm 1 returns the result obtained by the first core in this ordering who answered the satisfiability of the formula (lines 4-6).

This is a necessary but not a sufficient condition for the reproducibility of the results. To achieve a complete determinization of the parallel solver, let us take a closer look to the DPLL search engine associated to each core (Algorithm 2). In addition to the usual description of the main component of DPLL based SAT solvers, we can see that two successive synchronization barriers (<*barrier*>, lines 13 and 18) are added to the algorithm. To understand the role of these synchronizing points, we need to note both their placement inside the algorithm and the content of the region circumscribed by these two barriers. First, the barrier labeled $barrier_1$ (line 13) is placed just before any thread can return a final statement about the satisfiability of the tested CNF. This barrier prevents cores from

Algorithm 2: search(core$_i$)

```
   Data: A CNF formula F;
   Result: answer[i] = true if F is satisfiable; false if F is unsatisfiable, unknown otherwise
 1 begin
 2     nbConflicts=0;
 3     while (true) do
 4         if (!propagate()) then
 5             nbConflicts++;
 6             if (topLevel) then
 7                 answer[i]= false;
 8                 goto barrier_1;
 9             learntClause=analyze();
10             exportExtraClause(learntClause);
11             backtrack();
12             if (nbConflicts % period == 0) then
13                 barrier_1: <barrier>
14                 if (∃j|answer[j]! = unknown) then
15                     return answer[i];
16                 updatePeriod();
17                 importExtraClauses();
18                 <barrier>
19         else
20             if (!decide()) then
21                 answer[i]= true;
22                 goto barrier_1;
```

returning their solution (i.e. model or refutation proof) in an anarchic way, and forces them to wait for each other before stating the satisfiability of the formula (line 14 and 15). This is why the search engine of each core goes to the first barrier (labeled $barrier_1$) when the unsatisfiability is proved (backtrack to the top level of the search tree, lines 6-8), or when a model is found (lines 20-22). At line 14, if the satisfiability of the formula is answered by one of the cores ($\exists j|answer[j]! = unknown$), the algorithm returns its own $answer[i]$. If no thread can return a definitive answer, they all share information by importing conflict clauses generated by the other cores during the last period (line 17). After each one of them has finished to import clauses (second barrier, line 18), they continue to explore the search space looking for a solution. This second synchronization barrier is integrated to prevent each core from leaving the synchronization region before the others. In other words, when a given core enter this second barrier, it waits for all other cores until all of them have finished importing the foreign clauses. As different clauses ordering will induce different unit propagation ordering and consequently different search trees, the clauses learnt by the other cores are imported (line 17) while following a fixed order of the cores w.r.t. their thread ID.

To complete this detailed description, let us just specify the usual functions of the search engine. First, the *propagate*() function (line 4) applies classical unit propagation and returns *false* if a conflict occurs, and *true* otherwise. In the first case, a clause is learnt by the function *analyze*() (line 9), such a clause is added to the formula and exported to the other cores (line 10, *exportExtraClause*() function). These learned clauses are periodically imported in the synchronization region (line 17). In the second case, the *decide*() function

choses the next decision variable, assigns it and returns *true*, otherwise it returns *false* as all the variable are assigned i.e., a model is found.

Note that to maximize the dynamics of information exchange, each core can be synchronized with the other ones after each conflict, importing each learnt clause right after it has been generated. Unfortunately, this solution proves empirically inefficient, since a lot of time is wasted by the thread waiting. To avoid this problem, we propose to only synchronize the threads after some fixed number of conflicts *period* (line 10). This approach, called $(DP)^2LL\_static(period)$, does not update the period during search (no call to the function *updatePeriod*(), line 16). However, even if we have the "optimal" value of the parameter *period*, the problem of thread waiting at the synchronization barrier can not be completely eliminated. Indeed, as the different cores usually present different search behaviors (different search strategies, different size (i.e., number of clauses) of the learnt databases, etc.), using the same value of the *period* for all cores, leads inevitably to wasted waiting time at the first barrier.

2.2 Speed-based Dynamic Synchronization

In this section, we propose a speed-based dynamic synchronization of the value of the period. Our goal is to reduce as much as possible the time wasted by the different cores at the synchronization barrier. The time needed by each core to perform the same number of conflicts is difficult to estimate in advance; however we propose an interesting approximation measure that exploits the current state of the search engine. As decisions and unit propagations are two fundamental operations that dominate the SAT solver run time, estimating their cost might lead us to a better approximation of the progression speed of each solver. Consequently, our speed-based dynamic synchronization of the period is a function of the number of unit propagation.

Let us formally describe our approach. In our dynamic synchronization strategy, for each core or computing unit u_i, we consider a synchronization-time sequence as a set of steps t_i^k with $t_i^0 = 0$ and $t_i^k = t_i^{k-1} + period_i^k$ where $period_i^k$ represents the time window defined in term of number of conflicts between t_i^{k-1} and t_i^k. Obviously, this synchronization-time sequence is different for all the computing units u_i ($0 \leq i < nbCores$). Let us define Δ_i^k as the set of clauses currently in the learnt database of u_i at step t_i^k. In the sequel, when there is no ambiguity, we sometimes note t_i^k simply k.

Let us now formally describe the dynamic computation of these synchronization-time sequences. Let $m = max_{\forall u_i}(|\Delta_i^k|)$, where $0 \leq i < nbCores$, be the size of the largest learnt clauses database and $S_i^k = \frac{|\Delta_i^k|}{m}$ the ratio between the size of the learnt clauses database of u_i and m. This ratio S_i^k represents the speedup of u_i. When this ratio tends to one, the progression of the core u_i is closer to the slowest core, while when it tends to 0, the core u_i progresses more quickly than the slowest one. For $k = 0$ and for each u_i, we set $period_i^0$ to α, where α is a natural number. Then, at a given time step $k > 0$, and for each u_i, the next value of the period is computed as follows: $period_i^{k+1} = \alpha + (1 - S_i^k) \times \alpha$, where $0 \leq i < nbCores$. Intuitively, the core with the highest speedup S_i^k (tending to 1) should have the lowest period. On the contrary, the core with the lowest speedup S_i^k (tending to 0) should have the highest value of the period.

3. Evaluation

All the experimentations have been conducted on Intel Xeon 3GHz under Linux CentOS 4.1. (kernel 2.6.9) with a RAM limit of 2GB. Our deterministic DPLL algorithm has been implemented on top of the portfolio-based parallel solver `ManySAT` (version 1.1). The timeout was set to 900 seconds for each instance. We used the 100 instances proposed during the recent `SAT Race 2010`, and we report for each experiment the number of solved instances (x-axis) together with the total needed time (y-axis) to solve them. Each parallelized solver is running using 4 threads. Note that in the following experiments, we consider the real time used by the solvers, instead of the classic CPU time. Indeed, in most architectures, the CPU time is not increased when the threads are asleep (e.g. waiting time at the barriers), so taking the CPU time into account would give an illegitimate substantial advantage to $(DP)^2LL$.

3.1 Static Period

In a first experiment, we have evaluated the performance of our Deterministic Parallel DPLL ($(DP)^2$LL) with various static periods. Figure 1 presents the obtained results. First, a sequential version of the solver has been used (ManySAT using 1 core). Unsurprisingly, this version obtains the worst global results by only solving 68 instances in more than 11,000 seconds. This result enables to show the improvement obtained by the use of parallelized engines. We also report the results obtained by the non-deterministic solver `ManySAT 1.1`. Note that executing several times this version may lead to different results. This non-deterministic solver has been able to solve 75 instances within 8,850 seconds. Next, we ran a deterministic version of `ManySAT`, i.e., $(DP)^2LL$, where each core synchronizes with the other ones after each clause generation ($(DP)^2LL\_static(1)$). We can observe that the synchronization barrier is computationally expensive. Indeed, the deterministic version is clearly less efficient than the non-deterministic one, by only solving 72 instances in more than 10,000 seconds.

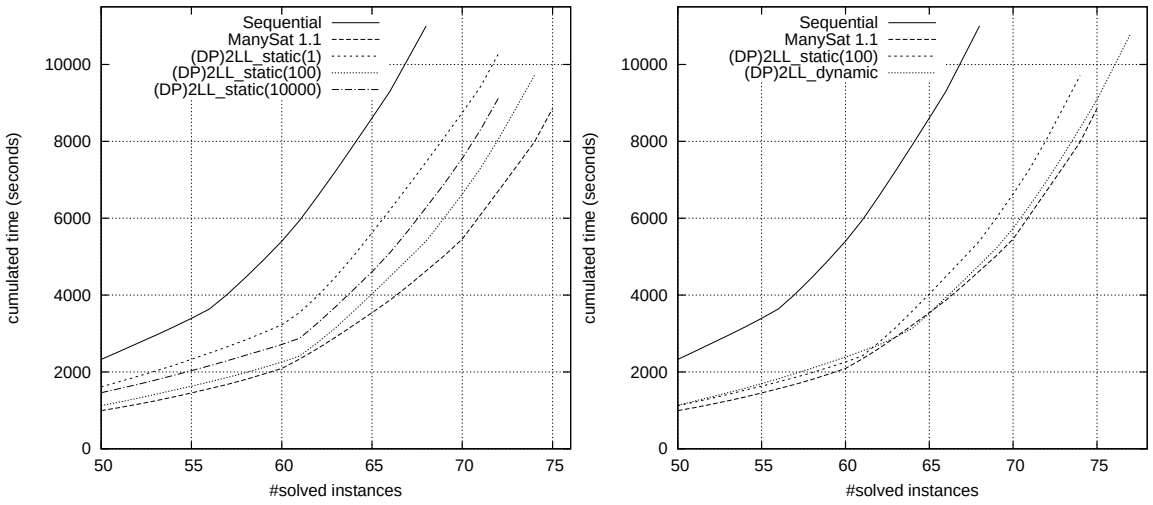

Figure 1. Performances using static and dynamic synchronizing

This negative result is mainly due to the time wasted by the cores waiting for each others on a (very) regular basis. To overcome this issue, we also tried to synchronize the different threads only after a given number of conflicts (100, 10000). Figure 1 shows that those versions outperform the "period=1" one, but still, stay far from the results obtained by the non-deterministic version.

3.2 Dynamic Period

In a second experiment, we tried to empirically evaluate our dynamic strategy. We compare the results of this version with the ones obtained by `ManySAT 1.1`, and with the results of the best static version (100), and of the sequential one too. The results are reported in Figure 1. The dynamic version is run with parameter $\alpha = 300$. This experiment empirically confirms the intuition that each core should have a different period value, w.r.t., the size of its own learnt clauses database, which heuristically indicates its unit propagation speed. Indeed, we can observe in Figure 1 that the "solving curve" of this dynamic version is really close to the one of `ManySat 1.1`. This means that the 2 solvers are able to solve about the same amount of instances within about the same time. Moreover, this adaptive version is able to solve 2 more instances than the non-deterministic one, which makes it the most efficient version tested during our experiments.

4. Discussion

In this paper, we have presented $(DP)^2LL$, the first deterministic parallelized procedure for SAT. This algorithm mainly consists in introducing two synchronization barriers to existing parallel portfolios. It has been integrated in ManySAT and its integration in other portfolios should be straight-forward.We have proposed and evaluated different synchronizing strategies and our experiments showed that our $(DP)^2LL$ can compete against a state-of-the-art non-deterministic parallel solver.

References

[1] A. Biere. Lingeling, plingeling, picosat and precosat at SAT race 2010. Technical Report 10/1, FMV Reports Series, 2010.

[2] Y. Hamadi, S. Jabbour, and L. Sais. ManySAT: a parallel SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation*, **6**:245–262, 2009.

[3] S. Kottler. SArTagnan: solver description. Technical report, SAT Race 2010, July 2010.

[4] T. Schubert, M. Lewis, and B. Becker. Antom: solver description. Technical report, SAT Race, 2010.